Site reliability through controlled disruption
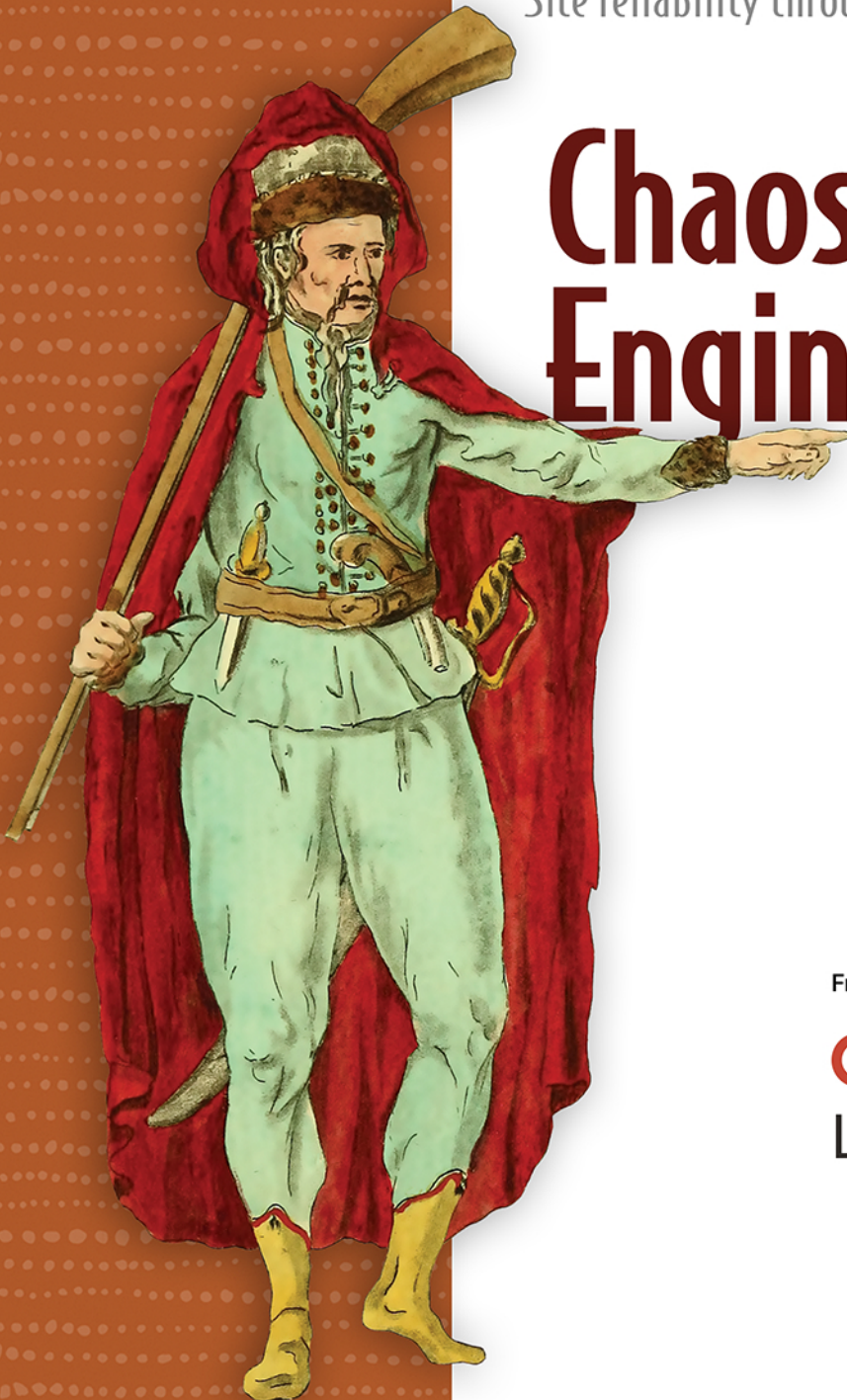
Chaos Engineering

Selected Chapters

Mikolaj Pawlikowski

From the team behind

ORACLE
Linux

Chaos Engineering
Site reliability through controlled disruption

Mikolaj Pawlikowski

Selected Chapters

contents

foreword

Netflix pioneered chaos engineering more than a decade ago and created a revolutionary testing approach—with tools such as Simian Army and Chaos Monkey—that helped make its distributed cloud-based systems more resilient.

Over that time, we also saw the evolution of distributed systems, hybrid and multicloud deployment environments, microservices and DevOps. The latter emphasizes continuous improvement and frequent releases of features and applications. This technology revolution in turn magnified cloud software instabilities, which ushered in the need to build ever more resiliency into cloud applications and infrastructure.

Your business may not be like Netflix but that doesn't mean you should overlook chaos engineering practices. One reason: downtime. How much does it cost your business in lost revenue, lower productivity, and reduced customer satisfaction? Whether the cost is measured in thousands or millions of dollars, it can be mitigated by utilizing chaos engineering practices.

It may seem counterintuitive for engineering to be preceded by chaos but when you plan and test for the unexpected, you can design more resiliency into what you engineer.

WHAT IS CHAOS ENGINEERING?

Wikipedia: Chaos engineering is the discipline of experimenting on a software system in production in order to build confidence in the system's capability to withstand turbulent and unexpected conditions.

Through the experimentation and testing practices of chaos engineering, you'll reduce the time, effort, and costs spent on incidents and the insights gleaned can help you improve system design. By continuously examining running software in production environments, application security and functionality can be enhanced. The result is improved resiliency and quality (= less downtime), which are important factors given the continued rise in distributed cloud-based systems and faster release cycles. For customers, fewer outages mean less disruption, which translates into improved service availability, durability, reliability, and satisfaction.

For these reasons, the Oracle Linux team has created this custom ebook, the first two chapters of Mikolaj Pawlikowski's *Chaos Engineering*. It is intended to help you

understand the movement behind chaos engineering and dispel concerns (it's really not that complicated). You will learn about the fundamentals and the four basic steps. You'll experience guided chaos experiments to help familiarize you with the concepts and a few pop quizzes will help reinforce your learning.

Mikolaj Pawlikowski has been practicing chaos engineering for four years, beginning with a large distributed Kubernetes-based microservices platform at Bloomberg. He is the creator of the Kubernetes Chaos Engineering tool PowerfulSeal, and the networking visibility tool Goldpinger. He is an active member of the Chaos Engineering community and speaks at numerous conferences.

Hand-in-hand with chaos engineering are the right tools to help you build your microservices-based, cloud native applications. For this, you'll want to explore Oracle Linux Cloud Native Environment. With Oracle Linux Cloud Native Environment, Oracle provides the components required by customers to develop microservices-based applications that can be deployed in environments that support open standards and specifications. Oracle closely tracks CNCF® and the Open Container Initiative projects and contributes to and abides by the standards defined by both. For example, Oracle Container Runtime for Docker is compliant with the Open Container Initiative image and runtime specifications and Oracle Linux Cloud Native Environment has achieved Certified Kubernetes status as defined by the CNCF.

Now, let's get on with *Chaos Engineering* and some monkey business.

Sergio Leunissen
Vice President, Development,
Oracle

Sergio Leunissen is a Vice President in Oracle's infrastructure engineering team. He joined Oracle in 1995 and has held a range of positions in sales engineering, development, and product management. Sergio currently leads initiatives to deliver solutions for developers on Oracle Linux and Oracle Cloud Infrastructure. He is also responsible for Oracle's presence on GitHub.

---

[1] Find more resources on

Chaos engineering: https://github.com/dastergon/awesome-chaos-engineering

Oracle Linux: oracle.com/linux

Oracle Linux Cloud Native Environment: https://blogs.oracle.com/linux/announcing-oracle-linux-cloud-native-environment-release-12

Oracle Linux Cloud Native Environment is a curated set of open source software selected from cloud native projects, integrated into a unified operating environment that includes:

- A rich set of software components for cloud native application development and deployment
- Certified Kubernetes and Kata Containers
- Enterprise-grade performance, scalability, reliability, and security

Oracle Linux Cloud Native Environment runs on-premises or in the cloud. It is tested and proven with Oracle products such as Oracle Database, Oracle WebLogic Server, MySQL and more.

To find out more see oracle.com/linux/cloudnative and oracle.com/linux

Stay connected:

- blogs.oracle.com/linux
- facebook.com/OracleLinux
- twitter.com/OracleLinux

about the author

Mikolaj Pawlikowski is a software engineer in love with reliability. Yup, "Miko" is fine!

If you'd like to hear more, join his newsletter at https://chaosengineering.news. To reach out directly, use LinkedIn or @mikopawlikowski on Twitter.

If you'd like to get involved in an open source chaos engineering project and hang out virtually, check out PowerfulSeal at https://github.com/powerfulseal/powerfulseal/.

And finally, Miko helps organize a yearly chaos engineering conference. Sign up at https://www.conf42.com.
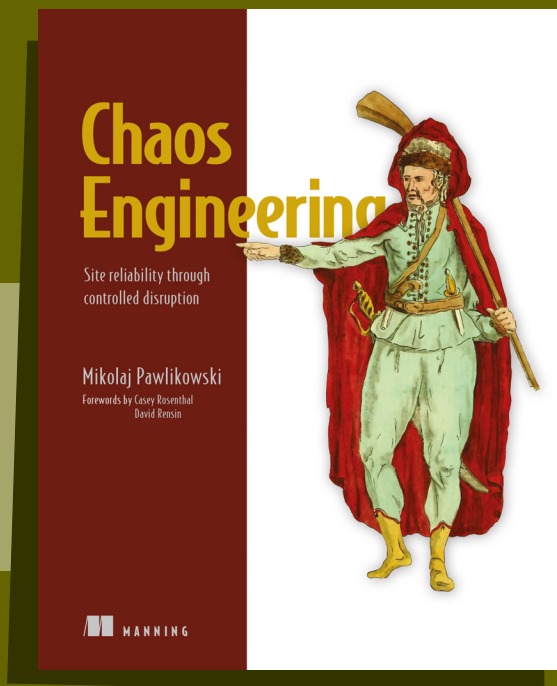
Into the world of
chaos engineering

This chapter covers

- What chaos engineering is and is not
- Motivations for doing chaos engineering
- Anatomy of a chaos experiment
- A simple example of chaos engineering in practice

What would you do to make absolutely sure the car you're designing is safe? A typical vehicle today is a real wonder of engineering. A plethora of subsystems, operating everything from rain-detecting wipers to life-saving airbags, all come together to not only go from A to B, but to protect passengers during an accident. Isn't it moving when your loyal car gives up the ghost to save yours through the strategic use of crumple zones, from which it will never recover?

Because passenger safety is the highest priority, all these parts go through rigorous testing. But even assuming they all work as designed, does that guarantee you'll survive in a real-world accident? If your business card reads, "New Car Assessment Program," you demonstrably don't think so. Presumably, that's why every new car making it to the market goes through crash tests.

1

Picture this: a production car, heading at a controlled speed, closely observed with high-speed cameras, in a lifelike scenario: crashing into an obstacle to test the system as a whole. In many ways, *chaos engineering is to software systems what crash tests are to the car industry*: a deliberate practice of experimentation designed to uncover systemic problems. In this book, you'll look at the why, when, and how of applying chaos engineering to improve your computer systems. And perhaps, who knows, save some lives in the process. What's a better place to start than a nuclear power plant?

1.1   What is chaos engineering?

Imagine you're responsible for designing the software operating a nuclear power plant. Your job description, among other things, is to prevent radioactive fallout. The stakes are high: a failure of your code can produce a disaster leaving people dead and rendering vast lands uninhabitable. You need to be ready for anything, from earthquakes, power cuts, floods, or hardware failures, to terrorist attacks. What do you do?

You hire the best programmers, set in place a rigorous review process, test coverage targets, and walk around the hall reminding everyone that we're doing serious business here. But "Yes, we have 100% test coverage, Mr. President!" will not fly at the next meeting. You need contingency plans; you need to be able to demonstrate that when bad things happen, the system as a whole can withstand them, and the name of your power plant stays out of the news headlines. You need to go looking for problems before they find you. That's what this book is about.

Chaos engineering is defined as "the discipline of experimenting on a system in order to build confidence in the system's capability to withstand turbulent conditions in production" (Principles of Chaos Engineering, http://principlesofchaos.org/). In other words, it's a software testing method focusing on finding evidence of problems before they are experienced by users.

You want your systems to be reliable (we'll look into that), and that's why you work hard to produce good-quality code and good test coverage. Yet, even if your code works as intended, in the real world plenty of things can (and will) go wrong. The list of things that can break is longer than a list of the possible side effects of painkillers: starting with sinister-sounding events like floods and earthquakes, which can take down entire datacenters, through power supply cuts, hardware failures, networking problems, resource starvation, race conditions, unexpected peaks of traffic, complex and unaccounted-for interactions between elements in your system, all the way to the evergreen operator (human) error. And the more sophisticated and complex your system, the more opportunities for problems to appear.

It's tempting to discard these as rare events, but they just keep happening. In 2019, for example, two crash landings occurred on the surface of the Moon: the Indian Chandrayaan-2 mission (http://mng.bz/Xd7v) and the Israeli Beresheet (http://mng .bz/yYgB), both lost on lunar descent. And remember that even if you do everything right, more often than not, you still depend on other systems, and these systems can

fail. For example, Google Cloud,[1] Cloudflare, Facebook (WhatsApp), and Apple all had major outages within about a month in the summer of 2019 (http://mng.bz/ d42X). If your software ran on Google Cloud or relied on Cloudflare for routing, you were potentially affected. That's just reality.

It's a common misconception that chaos engineering is only about randomly breaking things in production. It's not. Although running experiments in production is a unique part of chaos engineering (more on that later), it's about much more than that—anything that helps us be confident the system can withstand turbulence. It interfaces with site reliability engineering (SRE), application and systems performance analysis, and other forms of testing. Practicing chaos engineering can help you prepare for failure, and by doing that, learn to build better systems, improve existing ones, and make the world a safer place.

1.2 Motivations for chaos engineering

At the risk of sounding like an infomercial, there are at least three good reasons to implement chaos engineering:

- Determining risk and cost and setting service-level indicators, objectives, and agreements
- Testing a system (often complex and distributed) as a whole
- Finding emergent properties you were unaware of

Let's take a closer look at these motivations.

1.2.1 Estimating risk and cost, and setting SLIs, SLOs, and SLAs

You want your computer systems to run well, and the subjective definition of what *well* means depends on the nature of the system and your goals regarding it. Most of the time, the primary motivation for companies is to create profit for the owners and shareholders. The definition of *running well* will therefore be a derivative of the business model objectives.

Let's say you're working on a planet-scale website, called Bookface, for sharing photos of cats and toddlers and checking on your high-school ex. Your business model might be to serve your users targeted ads, in which case you will want to balance the total cost of running the system with the amount of money you can earn from selling these ads. From an engineering perspective, one of the main risks is that the entire site could go down, and you wouldn't be able to present ads and bring home the revenue. Conversely, not being able to display a particular cat picture in the rare event of a problem with the cat picture server is probably not a deal breaker, and will affect your bottom line in only a small way.

For both of these risks (users can't use the website, and users can't access a cat photo momentarily), you can estimate the associated cost, expressed in dollars per

---

[1] You can see the official, detailed Google Cloud report at http://mng.bz/BRMg.

unit of time. That cost includes the direct loss of business as well as various other, less tangible things like public image damage, that might be equally important. As a real-life example, Forbes estimated that Amazon lost $66,240 per minute of its website being down in 2013.[2]

Now, to quantify these risks, the industry uses *service-level indicators* (*SLIs*). In our example, the percentage of time that your users can access the website could be an SLI. And so could the ratio of requests that are successfully served by the cat photos service within a certain time window. The SLIs are there to put a number to an event, and picking the right SLI is important.

Two parties agreeing on a certain range of an SLI can form a *service-level objective* (*SLO*), a tangible target that the engineering team can work toward. SLOs, in turn, can be legally enforced as a *service-level agreement* (*SLA*), in which one party agrees to guarantee a certain SLO or otherwise pay some form of penalty if they fail to do so.

Going back to our cat- and toddler-photo-sharing website, one possible way to work out the risk, SLI, and SLO could look like this:

- The main risk is "People can't access the website," or simply the *downtime*
- A corresponding SLI could be "the ratio of success responses to errors from our servers"
- An SLO for the engineering team to work toward: "the ratio of success responses to errors from our servers > 99.95% on average monthly"

To give you a different example, imagine a financial trading platform, where people query an API when their algorithms want to buy or sell commodities on the global markets. Speed is critical. We could imagine a different set of constraints, set on the trading API:

- SLI: 99th percentile response time
- SLO: 99th percentile response time < 25 ms, 99.999% of the time

From the perspective of the engineering team, that sounds like mission impossible: we allow ourselves about only 5 minutes a year when the top 1% of the slowest requests average over 25 milliseconds (ms) response time. Building a system like that might be difficult and expensive.

Number of nines

In the context of SLOs, we often talk about the *number of nines* to mean specific percentages. For example, 99% is *two nines*, 99.9% is *three nines*, 99.999% is *five nines*, and so on. Sometimes, we also use phrases like *three nines five* or *three and a half nines* to mean 99.95%, although the latter is not technically correct (going from

2   See "Amazon.com Goes Down, Loses $66,240 per Minute," by Kelly Clay, Forbes, August 2013, http://mng .bz/ryJZ.

99.9% to 99.95% is a factor of 2, but going from 99.9% to 99.99% is a factor of 5). The following are a few of the most common values and their corresponding downtimes per year and per day:

- 90% (*one nine*)—36.53 days per year, or 2.4 hours per day
- 99% (*two nines*)—3.65 days per year, or 14.40 minutes per day
- 99.95% (*three and a half nines*)—4.38 hours per year, or 43.20 seconds per day
- 99.999% (*five nines*)—5.26 minutes per year, or 840 milliseconds per day

How does chaos engineering help with these? To satisfy the SLOs, you'll engineer the system in a certain way. You will need to take into account the various sinister scenarios, and the best way to see whether the system works fine in these conditions is to go and create them—which is exactly what chaos engineering is about! You're effectively working backward from the business goals, to an engineering-friendly defined SLO, that you can, in turn, continuously test against by using chaos engineering. Notice that in all of the preceding examples, I am talking in terms of entire systems.

1.2.2 Testing a system as a whole

Various testing techniques approach software at different levels. *Unit tests* typically cover single functions or smaller modules in isolation. *End-to-end* (*e2e*) *tests* and *integration tests* work on a higher level; whole components are put together to mimic a real system, and verification is done to ensure that the system does what it should. *Benchmarking* is yet another form of testing, focused on the performance of a piece of code, which can be lower level (for example, micro-benchmarking a single function) or a whole system (for example, simulating client calls).

I like to think of chaos engineering as the next logical step—a little bit like e2e testing, but during which we rig the conditions to introduce the type of failure we expect to see, and measure that we still get the correct answer within the expected time frame. It's also worth noting, as you'll see in part 2, that even a single-process system can be tested using chaos engineering techniques, and sometimes that comes in really handy.

1.2.3 Finding emergent properties

Our complex systems often exhibit *emergent properties* that we didn't initially intend. A real-world example of an emergent property is a human heart: its single cells don't have the property of pumping blood, but the right configuration of cells produces a heart that keeps us alive. In the same way, our neurons don't think, but their interconnected collection that we call a *brain* does, as you're illustrating by reading these lines.

In computer systems, properties often emerge from the interactions among the moving parts that the system comprises. Let's consider an example. Imagine that you run a system with many services, all using a Domain Name System (DNS) server to

find one another. Each service is designed to handle DNS errors by retrying up to 10 times. Similarly, the external users of the systems are told to retry if their requests ever fail. Now, imagine that, for whatever reason, the DNS server fails and restarts. When it comes back up, it sees an amount of traffic amplified by the layers of retries, an amount that it wasn't set up to handle. So it might fail again, and get stuck in an infinite loop restarting, while the system as a whole is down. No component of the system has the property of creating infinite downtime, but with the components together and the right timing of events, the system as a whole might go into that state.

Although certainly less exciting than the example of consciousness I mentioned before, this property emerging from the interactions among the parts of the system is a real problem to deal with. This kind of unexpected behavior can have serious consequences on any system, especially a large one. The good news is that chaos engineering excels at finding issues like this. By experimenting on real systems, often you can discover how simple, predictable failures can cascade into large problems. And once you know about them, you can fix them.

> ### Chaos engineering and randomness
> When doing chaos engineering, you can often use the element of randomness and borrow from the practice of *fuzzing*—feeding pseudorandom payloads to a piece of software in order to try to come up with an error that your purposely written tests might be missing. The randomness definitely can be helpful, but once again, I would like to stress that controlling the experiments is necessary to be able to understand the results; chaos engineering is not just about randomly breaking things.

Hopefully, I've had your curiosity and now I've got your attention. Let's see how to do chaos engineering!

1.3   *Four steps to chaos engineering*

Chaos engineering experiments (*chaos experiments*, for short) are the basic units of chaos engineering. You do chaos engineering through a series of chaos experiments. Given a computer system and a certain number of characteristics you are interested in, you design experiments to see how the system fares when bad things happen. In each experiment, you focus on proving or refuting your assumptions about how the system will be affected by a certain condition.

For example, imagine you are running a popular website and you own an entire datacenter. You need your website to survive power cuts, so you make sure two independent power sources are installed in the datacenter. In theory, you are covered—but in practice, a lot can still go wrong. Perhaps the automatic switching between power sources doesn't work. Or maybe your website has grown since the launch of the datacenter, and a single power source no longer provides enough electricity for all the servers. Did you remember to pay an electrician for a regular checkup of the machines every three months?

If you feel worried, you should. Fortunately, chaos engineering can help you sleep better. You can design a simple chaos experiment that will scientifically tell you what happens when one of the power supplies goes down (for more dramatic effect, always pick the newest intern to run these steps).

Repeat for all power sources, one at a time:

1 Check that The Website is up.
2 Open the electrical panel and turn the power source off.
3 Check that The Website is still up.
4 Turn the power source back on.

This process is crude, and sounds obvious, but let's review these steps. Given a computer system (a datacenter) and a characteristic (survives a single power source failure), you designed an experiment (switch a power source off and eyeball whether The Website is still up) that increases your confidence in the system withstanding a power problem. You used science for the good, and it took only a minute to set up. *That's one small step for man, one giant leap for mankind.*

Before you pat yourself on the back, though, it's worth asking what would happen if the experiment failed and the datacenter went down. In this overly-crude-for-demonstration-purposes case, you would create an outage of your own. A big part of your job will be about minimizing the risks coming from your experiments and choosing the right environment to execute them. More on that later.

Take a look at figure 1.1, which summarizes the process you just went through. When you're back, let me anticipate your first question: What if you are dealing with more-complex problems?
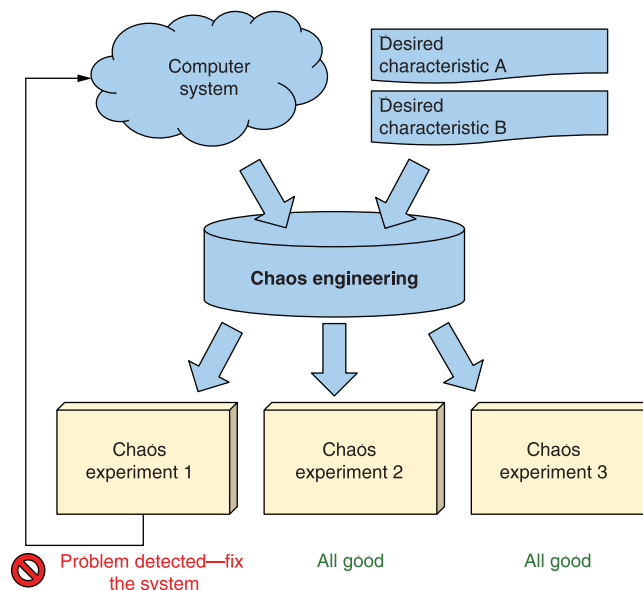

Figure 1.1 The process of doing chaos engineering through a series of chaos experiments

As with any experiment, you start by forming a hypothesis that you want to prove or disprove, and then you design the entire experience around that idea. When Gregor Mendel had an intuition about the laws of heredity, he designed a series of experiments on yellow and green peas, proving the existence of dominant and recessive traits. His results didn't follow the expectations, and that's perfectly fine; in fact, that's how his breakthrough in genetics was made.[3] We will be drawing inspiration from his experiments throughout the book, but before we get into the details of good craftsmanship in designing our experiments, let's plant a seed of an idea about what we're looking for.

Let's zoom in on one of these chaos experiment boxes from figure 1.1, and see what it's made of. Let me guide you through figure 1.2, which describes the simple, four-step process to design an experiment like that:

1. You need to be able to observe your results. Whether it's the color of the resulting peas, the crash test dummy having all limbs in place, your website being up, the CPU load, the number of requests per second, or the latency of successful requests, the first step is to ensure that you can accurately read the value for these
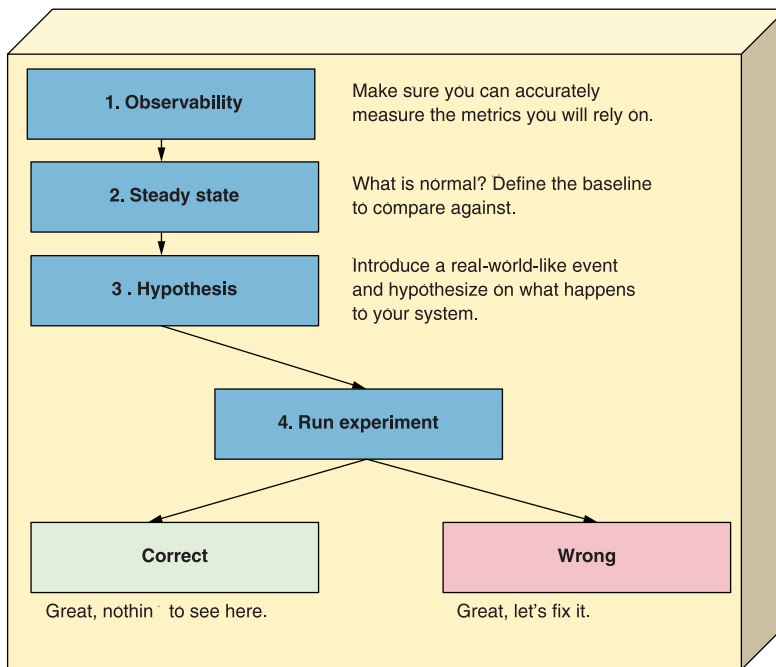

Figure 1.2   The four steps of a chaos experiment

---

[3] He did have to wait a couple of decades for anyone to reproduce his findings and for mainstream science to appreciate it and mark it "a breakthrough." But let's ignore that for now.

variables. We're lucky to be dealing with computers in the sense that we can often produce very accurate and very detailed data easily. We will call this *observability*.

2 Using the data you observe, you need to define what's *normal*. This is so that you can understand when things are out of the expected range. For instance, you might expect the CPU load on a 15-minute average to be below 20% for your application servers during the working week. Or you might expect 500 to 700 requests per second per instance of your application server running with four cores on your reference hardware specification. This normal range is often referred to as the *steady state.*

3 You shape your intuition into a hypothesis that can be proved or refuted, using the data you can reliably gather (observability). A simple example could be "Killing one of the machines doesn't affect the average service latency."

4 You execute the experiment, making your measurements to conclude whether you were right. And funnily enough, you like being wrong, because that's what you learn more from. Rinse and repeat.

The simpler your experiment, usually the better. You earn no bonus points for elaborate designs, unless that's the best way of proving the hypothesis. Look at figure 1.2 again, and let's dive just a little bit deeper, starting with *observability*.

1.3.1 *Ensure observability*

I quite like the word *observability* because it's straight to the point. It means being able to reliably see whatever metric you are interested in. The keyword here is *reliably*. Working with computers, we are often spoiled—the hardware producer or the operating system (OS) already provides mechanisms for reading various metrics, from the temperature of CPUs, to the fan's RPMs, to memory usage and hooks to use for various kernel events. But at the same time, it's often easy to forget that these metrics are subject to bugs and caveats that the end user needs to take into account. If the process you're using to measure CPU load ends up using more CPU than your application, that's probably a problem.

If you've ever seen a crash test on television, you will know it's both frightening and mesmerizing at the same time. Watching a 3000-pound machine accelerate to a carefully controlled speed and then fold like an origami swan on impact with a massive block of concrete is . . . humbling.

But the high-definition, slow-motion footage of shattered glass flying around, and seemingly unharmed (and unfazed) dummies sitting in what used to be a car just seconds before is not just for entertainment. Like any scientist who earned their white coat (and hair), both crash-test specialists and chaos engineering practitioners alike need reliable data to conclude whether an experiment worked. That's why observability, or reliably harvesting data about a live system, is paramount.

In this book, we're going to focus on Linux and the system metrics that it offers to us (CPU load, RAM usage, I/O speeds) as well as go through examples of higher-level metrics from the applications we'll be experimenting on.

> **Observability in the quantum realm**
>
> If your youth was as filled with wild parties as mine, you might be familiar with the double-slit experiment (http://mng.bz/MX4W). It's one of my favorite experiments in physics, and one that displays the probabilistic nature of quantum mechanics. It's also one that has been perfected over the last 200 years by generations of physicists.
>
> The experiment in its modern form consists of shooting photons (or matter particles such as electrons) at a barrier that has two parallel slits, and then observing what landed on the screen on the other side. The fun part is that if you don't observe which slit the particles go through, they behave like a wave and interfere with each other, forming a pattern on the screen. But if you try to detect (observe) which slit each particle went through, the particles will not behave like a wave. So much for reliable observability in quantum mechanics!

1.3.2   *Define a steady state*

Armed with reliable data from the previous step (observability), you need to define what's normal so that you can measure abnormalities. A fancier way of saying that is to *define a steady state*, which works much better at dinner parties.

What you measure will depend on the system and your goals about it. It could be "undamaged car going straight at 60 mph" or perhaps "99% of our users can access our API in under 200ms." Often, this will be driven directly by the business strategy.

It's important to mention that on a modern Linux server, a lot of things will be going on, and you're going to try your best to isolate as many variables as possible. Let's take the example of CPU usage of your process. It sounds simple, but in practice, a lot of things can affect your reading. Is your process getting enough CPU, or is it being stolen by other processes (perhaps it's a shared machine, or maybe a cron job updating the system kicked in during your experiment)? Did the kernel schedule allocate cycles to another process with higher priority? Are you in a virtual machine, and perhaps the hypervisor decided something else needed the CPU more?

You can go deep down the rabbit hole. The good news is that often you are going to repeat your experiments many times, and some of the other variables will be brought to light, but remembering that all these other factors can affect your experiments is something you should keep in the back of your mind.

1.3.3   *Form a hypothesis*

Now, for the really fun part. In step 3, you shape your intuitions into a testable hypothesis—an educated guess of what will happen to your system in the presence of a well-defined problem. Will it carry on working? Will it slow down? By how much?

In real life, these questions will often be prompted by incidents (unprompted problems you discover when things stop working), but the better you are at this game, the more you can (and should) preempt. Earlier in the chapter, I listed a few examples of what tends to go wrong. These events can be broadly categorized as follows:

- External events (earthquakes, floods, fires, power cuts, and so on)
- Hardware failures (disks, CPUs, switches, cables, power supplies, and so on)
- Resource starvation (CPU, RAM, swap, disk, network)
- Software bugs (infinite loops, crashes, hacks)
- Unsupervised bottlenecks
- Unpredicted emergent properties of the system
- Virtual machine (Java Virtual Machine, V8, others)
- Hardware bugs
- Human error (pushing the wrong button, sending the wrong config, pulling the wrong cable, and so forth)

We will look into how to simulate these problems as we go through the concrete examples in part 2 of the book. Some of them are easy (switch off a machine to simulate machine failure or take out the Ethernet cable to simulate network issues), while others will be much more advanced (add latency to a system call). The choice of failures to take into account requires a good understanding of the system you are working on.

Here are a few examples of what a hypothesis could look like:

- On frontal collision at 60 mph, no dummies will be squashed.
- If both parent peas are yellow, all the offspring will be yellow.
- If we take 30% of our servers down, the API continues to serve the 99th percentile of requests in under 200 ms.
- If one of our database servers goes down, we continue meeting our SLO.

Now, it's time to run the experiment.

1.3.4 Run the experiment and prove (or refute) your hypothesis

Finally, you run the experiment, measure the results, and conclude whether you were right. Remember, being wrong is fine—and much more exciting at this stage!

Everybody gets a medal in the following conditions:

- If you were right, congratulations! You just gained more confidence in your system withstanding a stormy day.
- If you were wrong, congratulations! You just found a problem in your system before your clients did, and you can still fix it before anyone gets hurt!

We'll spend some time on the good craftsmanship rules in the following chapters, including automation, managing the blast radius, and testing in production. For now, just remember that as long as this is good science, you learn something from each experiment.

1.4 What chaos engineering is not

If you're just skimming this book in a store, hopefully you've already gotten some value out of it. More information is coming, so don't put it away! As is often the case, the devil is in the details, and in the coming chapters you'll see in greater depth how

to execute the preceding four steps. I hope that by now you can clearly see the benefits of what chaos engineering has to offer, and roughly what's involved in getting to it.

But before we proceed, I'd like to make sure that you also understand what *not* to expect from these pages. Chaos engineering is not a silver bullet, and doesn't automatically fix your system, cure cancer, or guarantee weight loss. In fact, it might not even be applicable to your use case or project.

A common misconception is that chaos engineering is about randomly destroying stuff. I guess the name kind of hints at it, and Chaos Monkey (https://netflix.github.io/chaosmonkey/), the first tool to gain internet fame in the domain, relies on randomness quite a lot. But although randomness can be a powerful tool, and sometimes overlaps with fuzzing, you want to control the variables you are interacting with as closely as possible. More often than not, adding failure is the easy part; the hard part is to know where to inject it and why.

Chaos engineering is not just Chaos Monkey, Chaos Toolkit (https://chaostoolkit.org/), PowerfulSeal (https://github.com/bloomberg/powerfulseal) or any of the numerous projects available on GitHub. These are tools making it easier to implement certain types of experiments, but the real difficulty is in learning how to look critically at systems and predict where the fragile points might be.

It's important to understand that chaos engineering doesn't replace other testing methods, such as unit or integration tests. Instead, it complements them: just as airbags are tested in isolation, and then again with the rest of the car during a crash test, chaos experiments operate on a different level and test the system as a whole.

This book will not give you ready-made answers on how to fix your systems. Instead, it will teach you how to find problems by yourself and where to look for them. Every system is different, and although we'll look at common scenarios and gotchas together, you'll need a deep understanding of your system's weak spots to come up with useful chaos experiments. In other words, the value you get out of the chaos experiments is going to depend on your system, how well you understand it, how deep you want to go testing it, and how well you set up your observability shop.

Although chaos engineering is unique in that it can be applied to production systems, that's not the only scenario that it caters to. A lot of content on the internet appears to be centered around "breaking things in production," quite possibly because it's the most radical thing you can do, but, again, that's not all chaos engineering is about—or even its main focus. A lot of value can be derived from applying chaos engineering principles and running experiments in other environments too.

Finally, although some overlap exists, chaos engineering doesn't stem from chaos theory in mathematics and physics. I know: bummer. Might be an awkward question to answer at a family reunion, so better be prepared.

With these caveats out of the way, let's get a taste of what chaos engineering is like with a small case study.

1.5 A taste of chaos engineering

Before things get technical, let's close our eyes and take a quick detour to Glanden, a fictional island country in northern Europe. Life is enjoyable for Glandeners. The geographical position provides a mild climate and a prosperous economy for its hard-working people. At the heart of Glanden is Donlon, the capital with a large population of about 8 million people, all with a rich heritage from all over the world—a true cultural melting pot. It's in Donlon that our fictitious startup FizzBuzzAAS tries really hard to *make the world a better place.*

1.5.1 FizzBuzz as a service

FizzBuzzAAS Ltd. is a rising star in Donlon's booming tech scene. Started just a year ago, it has already established itself as a clear leader in the market of FizzBuzz as a Service. Recently supported by serious venture capital (VC) dollars, the company is looking to expand its market reach and scale its operations. The competition, exemplified by FizzBuzzEnterpriseEdition (https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition) is fierce and unforgiving. The FizzBuzzAAS business model is straightforward: clients pay a flat monthly subscription fee to access the cutting-edge APIs.

Betty, head of sales at FizzBuzzAAS, is a natural. She's about to land a big contract that could make or break the ambitious startup. Everyone has been talking about that contract at the water cooler for weeks. The tension is sky-high.

Suddenly, the phone rings, and everyone goes silent. It's the Big Company calling. Betty picks up. "Mhm . . . Yes. I understand." It's so quiet you can hear the birds chirping outside. "Yes ma'am. Yes, I'll call you back. Thank you."

Betty stands up, realizing everyone is holding their breath. "Our biggest client can't access the API."

1.5.2 A long, dark night

It was the first time in the history of the company that the entire engineering team (Alice and Bob) pulled an all-nighter. Initially, nothing made sense. They could successfully connect to each of the servers, the servers were reporting as healthy, and the expected processes were running and responding—so where did the errors come from?

Moreover, their architecture really wasn't that sophisticated. An external request would hit a load balancer, which would route to one of the two instances of the API server, which would consult a cache to either serve a precomputed response, if it was fresh enough, or compute a new one and store it in cache. You can see this simple architecture in figure 1.3.

Finally, a couple of gallons of coffee into the night, Alice found the first piece of the puzzle. "It's kinda weird," she said as she was browsing through the logs of one of the API server instances, "I don't see any errors, but all of these requests seem to stop at the cache lookup." Eureka! It wasn't long after that moment that she found the problem: their code gracefully handled the cache being down (connection refused, no host, and so on), but didn't have any time-outs in case of no response. It was downhill
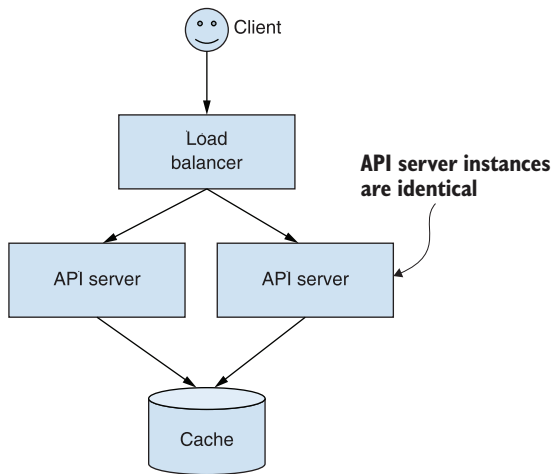
API server instances
are identical

Figure 1.3   FizzBuzz as a Service
technical architecture

from there—a quick session of pair programming, a rapid build and deploy, and it was time for a nap.

The order of the world was restored; people could continue requesting FizzBuzz as a Service, and the VC dollars were being well spent. The Big Company acknowledged the fix and didn't even mention cancelling its contract. The sun shone again. Later, it turned out that the API server's inability to connect to the cache was a result of a badly rolled-out firewall policy, in which someone forgot to whitelist the cache. Human error.

1.5.3   *Postmortem*

"How can we make sure that we're immune to this kind of issue the next time?" Alice asked, in what was destined to be a crucial meeting for the company's future.

Silence.

"Well, I guess we could preemptively set some of our servers on fire once in a while" answered Bob to lift up the mood just a little bit.

Everyone started laughing. Everyone, apart from Alice, that is.

"Bob, you're a genius!" Alice acclaimed and then took a moment to appreciate the size of everyone's eyeballs. "Let's do exactly that! If we could *simulate* a broken firewall rule like this, then we could add this to our integration tests."

"You're right!" Bob jumped out of his chair. "It's easy! I do it all the time to block my teenager's Counter Strike servers on the router at home! All you need to do is this," he said and proceeded to write on the whiteboard:

```
iptables -A ${CACHE_SERVER_IP} -j DROP
```

"And then after the test, we can undo that with this," he carried on, sensing the growing respect his colleagues were about to kindle in themselves:

```
iptables -D ${CACHE_SERVER_IP} -j DROP
```

Alice and Bob implemented these fixes as part of the setup and teardown of their integration tests, and then confirmed that the older version wasn't working, but the newer one including the fix worked like a charm. Both Alice and Bob changed their job titles to site reliability engineer (SRE) on LinkedIn the same night, and made a pact to never tell anyone they hot-fixed the issue in production.

1.5.4 Chaos engineering in a nutshell

If you've ever worked for a startup, long, coffee-fueled nights like this are probably no stranger to you. Raise your hand if you can relate! Although simplistic, this scenario shows all four of the previously covered steps in action:

- The *observability* metric is whether or not we can successfully call the API.
- The *steady state* is that the API responds successfully.
- The *hypothesis* is that if we drop connectivity to the cache, we continue getting a successful response.
- After *running the experiment*, we can confirm that the old version breaks and the new one works.

Well done, team: you've just increased confidence in the system surviving difficult conditions! In this scenario, the team was reactive; Alice and Bob came up with this new test only to account for an error their users already noticed. That made for a more dramatic effect on the plot. In real life, and in this book, we're going to do our best to predict and proactively detect this kind of issue without the external stimulus of becoming jobless overnight! And I promise that we'll have some serious fun in the process (see appendix D for a taste).

Summary

- Chaos engineering is a discipline of experimenting on a computer system in order to uncover problems, often undetected by other testing techniques.
- Much as the crash tests done in the automotive industry try to ensure that the car as a whole behaves in a certain way during a well-defined, real-life-like event, chaos engineering experiments aim to confirm or refute your hypotheses about the behavior of the system during a real-life-like problem.
- Chaos engineering doesn't automatically solve your issues, and coming up with meaningful hypotheses requires a certain level of expertise in the way your system works.
- Chaos engineering isn't about randomly breaking things (although that has its place, too), but about adding a controlled amount of failure you understand.
- Chaos engineering doesn't need to be complicated. The four steps we just covered, along with some good craftsmanship, should take you far before things get any more complex. As you will see, computer systems of any size and shape can benefit from chaos engineering.

First cup of chaos and blast radius

This chapter covers

- Setting up a virtual machine to run through accompanying code
- Using basic Linux forensics—why did your process die?
- Performing your first chaos experiment with a simple bash script
- Understanding the blast radius

The previous chapter covered what chaos engineering is and what a chaos experiment template looks like. It is now time to get your hands dirty and implement an experiment from scratch! I'm going to take you step by step through building your first chaos experiment, using nothing more than a few lines of bash. I'll also use the occasion to introduce and illustrate new concepts like *blast radius*.

 Just one last pit stop before we're off to our journey: let's set up the workspace.

DEFINITION   I'll bet you're wondering what a *blast radius* is. Let me explain. Much like an explosive, a software component can go wrong and break other things it connects to. We often speak of a blast radius to describe the maximum number of things that can be affected by something going wrong. I'll teach you more about it as you read this chapter.

2.1   *Setup: Working with the code in this book*

I care about your learning process. To make sure that all the relevant resources and tools are available to you immediately, I'm providing a virtual machine (VM) image that you can download, import, and run on any host capable of running VirtualBox. Throughout this book, I'm going to assume you are executing the code provided in the VM. This way, you won't have to worry about installing the various tools on your PC. It will also allow us to be more playful inside the VM than if it was your host OS.

Before you get started, you need to import the virtual machine image into Virtual-Box. To do that, complete the following steps:

1   Download the VM image:
   – Go to https://github.com/seeker89/chaos-engineering-book.
   – Click the Releases link at the right of the page.
   – Find the latest release.
   – Follow the release notes to download, verify, and decompress the VM archive (there will be multiple files to download).
2   Install VirtualBox by following instructions at www.virtualbox.org/wiki/Downloads.
3   Import the VM image into VirtualBox:
   – In VirtualBox, click File > Import Appliance.
   – Pick the VM image file you downloaded and unarchived.
   – Follow the wizard until completion.
4   Configure the VM to your taste (and resources):
   – In VirtualBox, right-click your new VM and choose Settings.
   – Choose General > Advanced > Shared Clipboard and then select Bidirectional.
   – Choose System > Motherboard and then select 4096 MB of Base Memory.
   – Choose Display > Video Memory and then select at least 64 MB.
   – Choose Display > Remote Display and then uncheck Enable Server.
   – Choose Display > Graphics Controller and then select what VirtualBox recommends.
5   Start the VM and log in.
   – The username and password are both `chaos`.

NOTE   When using VirtualBox, the Bidirectional check box under General > Advanced > Shared Clipboard activates copying and pasting in both directions. With this setting, you can copy things from your host machine by pressing Ctrl-C (Cmd-C on a Mac) and paste them into the VM with Ctrl-V (Cmd-V). A common gotcha is that when pasting into the terminal in Ubuntu, you need to press Ctrl-Shift-C and Ctrl-Shift-V.

That's it! The VM comes with all the source code needed and all the tools preinstalled. The versions of the tools will also match the ones I use in the text of this book.

All the source code, including the code used to prebuild the VM, can be found at https://github.com/seeker89/chaos-engineering-book. Once you've completed these steps, you should be able to follow everything in this book. If you find any issues, feel free to create an issue on that GitHub repo. Let's get to the meat of it by introducing an ironically realistic scenario!

> **TIP**   I chose VirtualBox because it's free and accessible to all. If you and VirtualBox don't get along, feel free to use the image with whatever does float your boat. VMware is a popular choice, and you can easily google how to set it up.

2.2   Scenario

Remember our friends from Glanden from the previous chapter? They have just reached out for help. They are having trouble with their latest product: the early clients are complaining it sometimes doesn't work, but when the engineers are testing, it all seems fine. As a growing authority in the chaos engineering community, you agree to help them track and fix the issue they are facing. Challenge accepted.

This is a more common scenario than any chaos engineer would like to admit. Something's not working, the existing testing methods don't find anything, and the clock is ticking. In an ideal world, you would proactively think about and prevent situations like this, but in the real world, you'll often face problems that are already there. To give you the right tools to cope, I want to start you off with a scenario of the latter category.

In this kind of situation, you'll typically have at least two pieces of information to work with: the overall architecture and the application logs. Let's start by taking a look at the architecture of the FizzBuzz service, shown in figure 2.1.

1. Client makes an HTTP request to the system

6. Load balancer returns the response. The failure in step 3 is invisible to the user.

2. Load balancer chooses an instance of the API server

3. Instance A becomes unavailable; request fails

5. Instance B successfully responds.

4. Load balancer tries instance B

Client

Load balancer

API server (instance A)
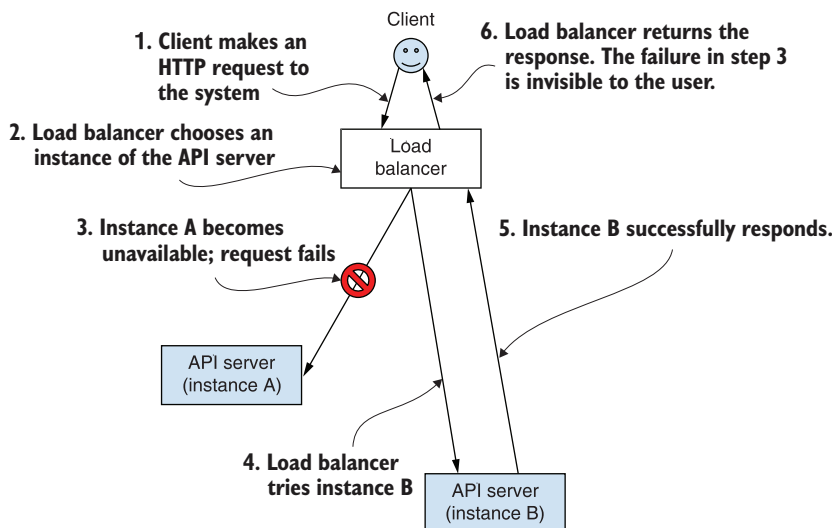
API server (instance B)

Figure 2.1   FizzBuzz as a Service technical architecture

As figure 2.1 illustrates, the architecture consists of a load balancer (NGINX) and two identical copies of an API server (implemented in Python). When a client makes a request through their internet browser (1), the request is received by the load balancer. The load balancer is configured to route incoming traffic to any instance that's up and running (2). If the instance the load balancer chooses becomes unavailable (3), the load balancer is configured to retransmit the request to the other instance (4). Finally, the load balancer returns the response provided by the instance of the API server to the client (5), and the internal failure is transparent to the user.

The other element you have at your disposal is the logs. A relevant sample of the logs looks like this (similar lines appear multiple times):

```
[14658.582809] ERROR: FizzBuzz API instance exiting, exit code 143
[14658.582809] Restarting
[14658.582813] FizzBuzz API version 0.0.7 is up and running.
```

While a little bit light on the details, it does provide valuable clues about what is going on: you can see that their API server instances are restarted and you can also see something called an exit code. These restarts are a good starting point for designing a chaos experiment. But before we do that, it's important that you know how to read exit codes like these and use them to understand what happened to a process before it died. With the *Criminal Minds* theme in the background, let's take a look at the basics of Linux forensics.

2.3   Linux forensics 101

When doing chaos engineering, you will often find yourself trying to understand why a program died. It often feels like playing detective, solving mysteries in a popular crime TV series. Let's put on the detective hat and solve a case!

In the preceding scenario, what you have at your disposal amounts to a *black-box* program that you can see died, and you want to figure out *why*. What do you do, and how do you check what happened? This section covers exit codes and killing processes, both manually through the `kill` command and by the Out-Of-Memory Killer, a part of Linux responsible for killing processes when the system runs low on memory. This will prepare you to deal with processes dying in real life. Let's begin with the exit codes.

> **DEFINITION**   In software engineering, we often refer to systems that are *opaque* to us as *black boxes*; we can see only their inputs and outputs, and not their inner workings. The opposite of a black box is sometimes called a *white box*. (You might have heard about the bright orange recording devices installed on airplanes. They are also often referred to as *black boxes*, because they are designed to prevent tampering with them, despite their real color.) When practicing chaos engineering, we will often be able to operate on entire systems or system components that are black boxes.

2.3.1   *Exit codes*

When dealing with a black-box piece of code, the first thing you might want to think about is running the program and seeing what happens. Unless it's supposed to rotate nuclear plant access codes, running it might be a good idea. To show you what that could look like, I wrote a program that dies. Let's warm up by running it and investigating what happens. From the provided VM, open a new bash session and start a mysterious program by running this command:

```
~/src/examples/killer-whiles/mystery000
```

You will notice that it exits immediately and prints an error message like this:

```
Floating point exception (core dumped)
```

The program was kind enough to tell us why it died: something to do with a floating-point arithmetic error. That's great for a human eye, but Linux provides a better mechanism for understanding what happened to the program. When a process terminates, it returns a number to inform the user of whether it succeeded. That number is called an *exit code.* You can check the exit code returned by the preceding command by running the following command at the prompt:

```
echo $?
```

In this case, you will see the following output:

```
136
```

This means that the last program that executed exited with code 136. Many (not all) UNIX commands return 0 when the command is successful, and 1 when it fails. Some use different return codes to differentiate various errors. Bash has a fairly compact convention on exit codes that I encourage you to take a look at (www.tldp.org/LDP/abs/html/exitcodes.html).

The codes in range 128–192 are decoded by using $128 + n$, where n is the number of the kill signal. In this example, the exit code is 136, which corresponds to $128 + 8$, meaning that the program received a kill signal number 8, which is SIGFPE. This signal is sent to a program when it tries to execute an erroneous arithmetic operation. Don't worry—you don't have to remember all the kill signal numbers by heart. You can see them with their corresponding numbers by running kill -L at the command prompt. Note that some of the exit codes differ between bash and other shells.

Remember that a program can return any exit code, sometimes by mistake. But assuming that it gives you a meaningful exit code, you know where to start debugging, and life tends to be good. The program did something wrong, it died, the cold kernel justice was served. But what happens if you suspect a murder?

Available signals

If you're curious about the various signals you can send (for example, via the `kill` command), you can list them easily by running the following command in your terminal:

```
kill -L
```

You will see output similar to the following:

```
 1)SIGHUP       2)SIGINT       3)SIGQUIT      4)SIGILL       5)SIGTRAP
 6)SIGABRT      7)SIGBUS       8)SIGFPE       9)SIGKILL     10)SIGUSR1
11)SIGSEGV     12)SIGUSR2     13)SIGPIPE     14)SIGALRM     15)SIGTERM
16)SIGSTKFLT   17)SIGCHLD     18)SIGCONT     19)SIGSTOP     20)SIGTSTP
21)SIGTTIN     22)SIGTTOU     23)SIGURG      24)SIGXCPU     25)SIGXFSZ
26)SIGVTALRM   27)SIGPROF     28)SIGWINCH    29)SIGIO       30)SIGPWR
31)SIGSYS      34)SIGRTMIN    35)SIGRTMIN+1  36)SIGRTMIN+2  37)SIGRTMIN+3
38)SIGRTMIN+4 39)SIGRTMIN+5 40)SIGRTMIN+6  41)SIGRTMIN+7  42)SIGRTMIN+8
43)SIGRTMIN+9  44)SIGRTMIN+10 45)SIGRTMIN+11 46)SIGRTMIN+12 47)SIGRTMIN+13
48)SIGRTMIN+14 49)SIGRTMIN+15 50)SIGRTMAX-14 51)SIGRTMAX-13 52)SIGRTMAX-12
53)SIGRTMAX-11 54)SIGRTMAX-10 55)SIGRTMAX-9  56)SIGRTMAX-8  57 SIGRTMAX-7
58)SIGRTMAX-6  59)SIGRTMAX-5  60)SIGRTMAX-4  61)SIGRTMAX-3  62)SIGRTMAX-2
63)SIGRTMAX-1  64)SIGRTMAX
```

2.3.2 *Killing processes*

To show you how to explicitly kill processes, let's play both the good cop and the bad cop. Open two bash sessions in two terminal windows. In the first one, run the following command to start a long-running process:

```
sleep 3600
```

Just as its name indicates, the `sleep` command blocks for the specified number of seconds. This is just to simulate a long-running process. Your prompt will be blocked, waiting for the command to finish. To confirm that the process is there, in the second terminal, run the following command to list the running processes (the `f` flag shows visually the parent-child relationships between processes):

```
ps f
```

In the following output, you can see `sleep 3600` as a child of the other bash process:

```
PID  TTY       STAT    TIME COMMAND
4214 pts/1     Ss      0:00 bash
4262 pts/1     R+      0:00  \_ ps f
2430 pts/0     Ss      0:00 bash
4261 pts/0     S+      0:00  \_ sleep 3600
```

Now, still in the second terminal, let's commit a process crime—kill your poor sleep process:

```
pkill sleep
```

You will notice the sleep process die in the first terminal. It will print this output, and the prompt will become available again:

```
Terminated
```

This is useful to see, but most of the time, the processes you care about will die when you're not looking at them, and you'll be interested in gathering as much information about the circumstances of their death as possible. That's when the exit codes we covered before become handy. You can verify what exit code the sleep process returned before dying by using this familiar command:

```
echo $?
```

The exit code is 143. Similar to 136, it corresponds to 128 + 15, or SIGTERM, the default signal sent by the kill command. This is the same code that was displayed in the FizzBuzz logs, giving us an indication that their processes were being killed. This is an aha moment: a first piece to our puzzle!

If you chose a different signal, you would see a different exit code. To illustrate that, start the sleep process again from the first terminal by running the same command:

```
sleep 3600
```

To send a KILL signal, run the following command from the second terminal:

```
pkill -9 sleep
```

This will result in getting a different exit code. To see the exit code, run this command from the first terminal, the one in which the process died:

```
echo $?
```

You will see the following output:

```
137
```

As you might expect, the exit code is 137, or 128 + 9. Note that nothing prevents us from using kill -8, and getting the same exit code as in the previous example that had an arithmetic error in the program. All of this is just a convention, but most of the popular tooling will follow it.

So now you've covered another popular way for a process to die, by an explicit signal. It might be an administrator issuing a command, it might be the system detecting an arithmetic error, or it might be done by some kind of daemon managing the process. Of the latter category, an interesting example is the Out-Of-Memory (OOM) Killer. Let's take a look at the mighty killer.

Pop quiz: Return codes

Pick the false statement:

1 Linux processes provide a number that indicates the reason for exiting.
2 Number 0 means a successful exit.
3 Number 143 corresponds to `SIGTERM`.
4 There are 32 possible exit codes.

See appendix A for answers.

2.3.3 *Out-Of-Memory Killer*

The OOM Killer can be a surprise the first time you learn about it. If you haven't yet, I'd like you to experience it firsthand. Let's start with a little mystery to solve. To illustrate what OOM is all about, run the following program I've prepared for you from the command line:

```
~/src/examples/killer-whiles/mystery001
```

Can you find out what the program is doing? Where would you start? The source code is in the same folder as the executable, but stay with me for a few minutes before you read it. Let's try to first approach it as a black box.

   After a minute or two of running the program, you might notice your VM getting a little sluggish, which is a good hint to check the memory utilization. You can see that by running the `top` command from the command line, as follows:

```
top -n1 -o+%MEM
```

Note the use of `-n1` flag to print one output and exit, rather than update continuously, and `-o+%MEM` to sort the processes by their memory utilization.

   Your output will be similar to the following:

Free memory at around 100 MB

```
top - 21:35:49 up  4:21,  1 user,  load average: 0.55, 0.46, 0.49
Tasks: 175 total,   3 running, 172 sleeping,   0 stopped,   0 zombie
%Cpu(s): 11.8 us, 29.4 sy, 0.0 ni, 35.3 id, 20.6 wa, 0.0 hi, 2.9 si, 0.0 st
MiB Mem :  3942.4 total,   98.9 free,   3745.5 used,    98.0 buff/cache   <─
MiB Swap:     0.0 total,    0.0 free,      0.0 used.    5.3 avail Mem

PID  USER    PR  NI    VIRT    RES    SHR  S  %CPU  %MEM  TIME+   COMMAND
5451 chaos   20   0  3017292  2.9g     0  S   0.0  74.7  0:07.95 mystery001  <─
5375 chaos   20   0  3319204 301960 50504 S  29.4   7.5  0:06.65 gnome-shell
1458 chaos   20   0   471964 110628 44780 S   0.0   2.7  0:42.32 Xorg
(...)
```

Memory usage (RES and %MEM) and the name of mystery001 process in bold font

You can see that `mystery001` is using 2.9 GB of memory, almost three-quarters for the VM, and the available memory hovers around 100 MB. Your `top` might start dying on

you or struggle to allocate memory. Unless you're busy encoding videos or maxing out games, that's rarely a good sign. While you're trying to figure out what's going on, if my timing is any good, you should see your process die in the prompt (if you're running your VM with more RAM, it might take longer):

```
Killed
```

A murder! But what happened? Who killed it? The title of this section is a little bit of a giveaway, so let's check the kernel log to look for clues. To do that, you can use `dmesg`. It's a Linux utility that displays kernel messages. Let's search for our `mystery001` by running the following in a terminal:

```
dmesg | grep -i mystery001
```

You will see something similar to the following output. As you read through these lines, the plot thickens. Something called `oom_reaper` just killed your mysterious process:

```
[14658.582932] Out of memory: Kill process 5451 (mystery001)
score 758 or sacrifice child
[14658.582939] Killed process 5451 (mystery001)
total-vm:3058268kB, anon-rss:3055776kB, file-rss:4kB, shmem-rss:0kB
[14658.644154] oom_reaper: reaped process 5451 (mystery001),
now anon-rss:0kB, file-rss:0kB, shmem-rss:0kB
```

What is that, and why is it claiming rights to your processes? If you browse through `dmesg` a bit more, you will see a little information about what OOM Killer did, including the list of processes it evaluated before sacrificing your program on the altar of RAM.

Here's an example, shortened for brevity. Notice the `oom_score_adj` column, which displays the scores of various processes from the OOM Killer's point of view (I put the name in bold for easier reading):

```
[14658.582809] Tasks state (memory values in pages):
[14658.582809] [pid ] uid tgid total_vm rss pgtables_bytes swapents
    oom_score_adj name
(...)
[14658.582912] [5451] 1000  5451 764567  763945  6164480  0   0 mystery001
(...)
[14658.582932] Out of memory: Kill process 5451 (mystery001) score 758 or
    sacrifice child
[14658.582939] Killed process 5451 (mystery001) total-vm:3058268kB, anon-
    rss:3055776kB, file-rss:4kB, shmem-rss:0kB
[14658.644154] oom_reaper: reaped process 5451 (mystery001), now anon-
    rss:0kB, file-rss:0kB, shmem-rss:0kB
```

The OOM Killer is one of the more interesting (and controversial) memory management features in the Linux kernel. Under low-memory conditions, the OOM Killer kicks in and tries to figure out which processes to kill in order to reclaim some memory and for the system to regain some stability. It uses heuristics (including `niceness`,

how recent the process is and how much memory it uses—see https://linux-mm.org/ OOM_Killer for more details) to score each process and pick the unlucky winner. If you're interested in how it came to be and why it was implemented the way it was, the best article on this subject that I know of is "Taming the OOM Killer" by Goldwyn Rodrigues (https://lwn.net/Articles/317814/).

So, there it is, the third popular reason for processes to die, one that often surprises newcomers. In the FizzBuzz logs sample, you know that the exit code you saw could be a result of either an explicit `kill` command or perhaps the OOM Killer. Unfortunately, unlike other exit codes that have a well-defined meaning, the one in the logs sample doesn't help you conclude the exact reason for the processes dying. Fortunately, chaos engineering allows you to make progress regardless of that. Let's go ahead and get busy applying some chaos engineering!

Pop quiz: What's OOM?

Pick one:

1 A mechanism regulating the amount of RAM any given process is given
2 A mechanism that kills processes when the system runs low on resources
3 A yoga mantra
4 The sound that Linux admins make when they see processes dying

See appendix A for answers.

OOM Killer settings

The OOM Killer behavior can be tweaked via flags exposed by the kernel. The following is from the kernel documentation, www.kernel.org/doc/Documentation/sysctl/vm.txt:

```
==============================================================

oom_kill_allocating_task

This enables or disables killing the OOM-triggering task in
out-of-memory situations.

If this is set to zero, the OOM killer will scan through the entire
tasklist and select a task based on heuristics to kill.  This normally
selects a rogue memory-hogging task that frees up a large amount of
memory when killed.

If this is set to non-zero, the OOM killer simply kills the task that
triggered the out-of-memory condition.  This avoids the expensive
tasklist scan.

If panic_on_oom is selected, it takes precedence over whatever value
is used in oom_kill_allocating_task.

The default value is 0.
```

In addition, `oom_dump_tasks` will dump extra information when killing a process for easier debugging. In the provided VM based off Ubuntu Disco Dingo, you can see both flags defaulting to `0` and `1`, respectively, meaning that the OOM Killer will attempt to use its heuristics to pick the victim and then dump extra information when killing processes. If you want to check the settings on your system, you can run the following commands:

```
cat /proc/sys/vm/oom_kill_allocating_task
cat /proc/sys/vm/oom_dump_tasks
```

2.4   *The first chaos experiment*

The exit codes from the logs didn't give you a good indication of what was causing FizzBuzz's API servers to die. While this might feel like an anticlimax, it is by design. Through that dead end, I want to lead you to a powerful aspect of chaos engineering: we work on hypotheses about the entire system as a whole.

As you'll recall (look at figure 2.2 for a refresher), the system is designed to handle API server instances dying through load balancing with automatic rerouting if one instance is down. Alas, the users are complaining that they are seeing errors!
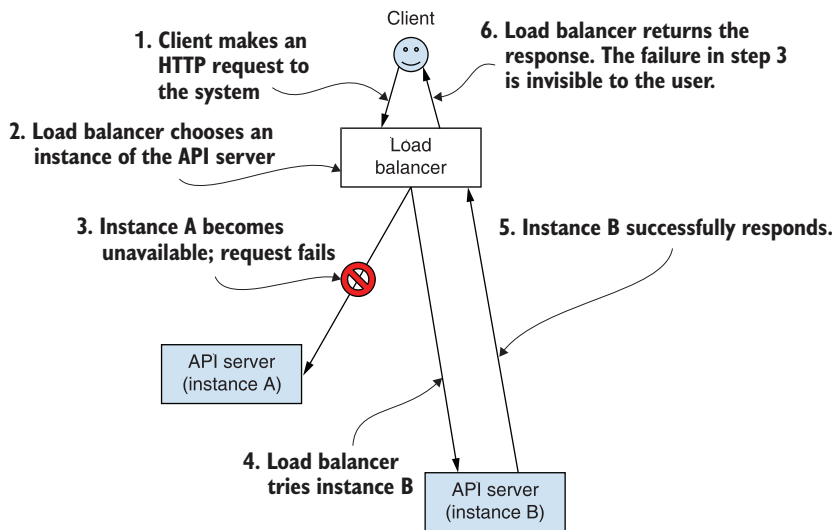

Figure 2.2   **FizzBuzz as a Service technical architecture, repeated**

While drilling down and fixing the reason that the API server instances get killed is important, from the perspective of the whole system, you should be more concerned that the clients are seeing the errors when they shouldn't. In other words, fixing the issue that gets the API server instances killed would "solve" the problem

for now, until another bug, outage, or human error reintroduces it, and the end users are impacted. In our system, or any bigger distributed system, components dying is a norm, not an exception.

Take a look at figure 2.3, which illustrates the difference in thinking about the system's properties as whole, as compared to figure 2.2. The client interacts with the system, and for just a minute we stop thinking about the implementation and think about how the system should behave as a single unit.
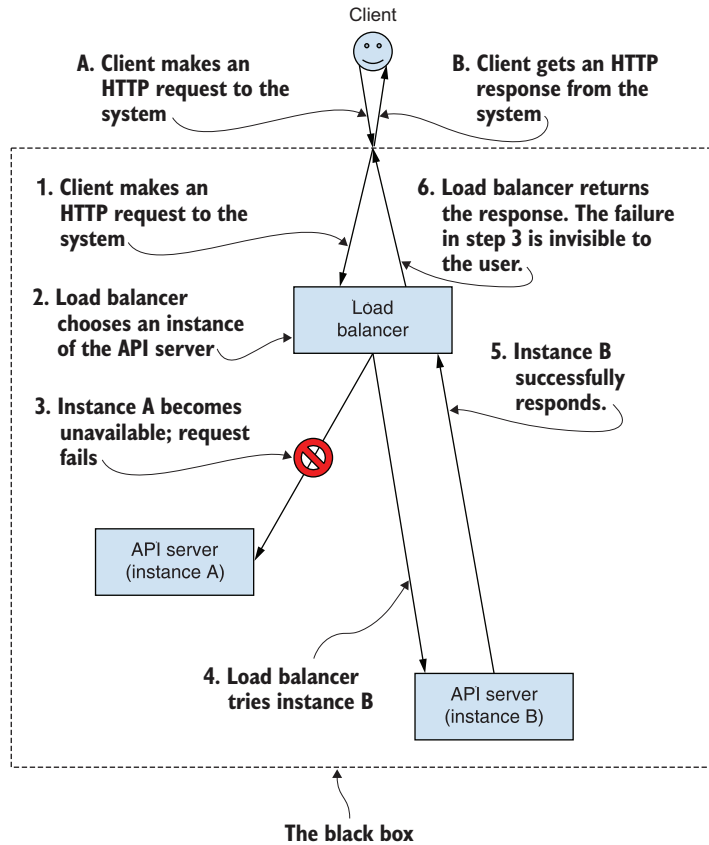
Client

A. Client makes an HTTP request to the system

B. Client gets an HTTP response from the system

1. Client makes an HTTP request to the system

6. Load balancer returns the response. The failure in step 3 is invisible to the user.

Load balancer

2. Load balancer chooses an instance of the API server

5. Instance B successfully responds.

3. Instance A becomes unavailable; request fails

API server (instance A)

4. Load balancer tries instance B

API server (instance B)

The black box

Figure 2.3   FizzBuzz as a Service whole system properties

Let's design our first chaos experiment to replicate the situation our clients are facing and see what happens for ourselves. The previous chapter presented the four steps to designing a chaos experiment:

1 Ensure observability.
2 Define a steady state.

3   Form a hypothesis.
4   Run the experiment!

It's best to start as simply as possible. You need a metric to work with (observability), preferably one you can produce easily. In this case, let's pick the number of failed HTTP responses that you receive from the system. You could write a script to make requests and count the failed ones for you, but existing tools can do that for you already.

To keep things simple, you'll use a tool that's well known: *Apache Bench*. You can use it to both produce the HTTP traffic for you to validate the steady state and to produce the statistics on the number of error responses encountered in the process. If the system behaves correctly, you should see no error responses, even if you kill an instance of the API server during the test. And that's going to be our hypothesis. Finally, implementing and running the experiment will also be simple, as we've just covered killing processes.

To sum it up, I've prepared figure 2.4, which should look familiar. It's the four-steps template from chapter 1, figure 1.2, with the details of our first experiment filled in. Please take a look.
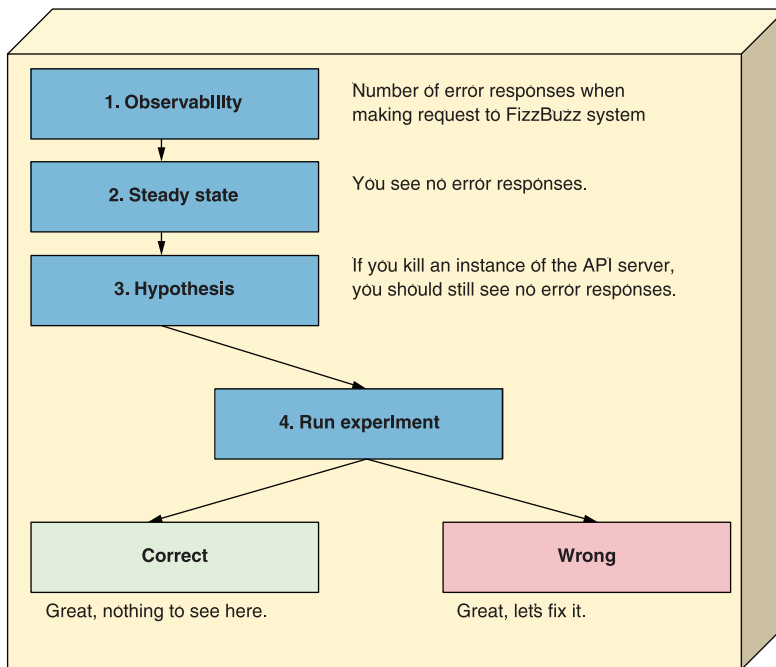
Figure 2.4   The four steps of our first chaos experiment

If this sounds like a plan to you, we're on the right track. It's finally time to get your hands dirty! Let's take a closer look at our application. Your VM comes with all the

components preinstalled and all the source code can be found in the ~/src/examples/killer_whiles folder. The two instances of the API server are modeled as systemd services faas001_a and faas001_b. They come preinstalled (but disabled by default), so you can use systemctl to check their status. Use the command prompt to run this command for either faas001_a or faas001_b (and press Q to exit):

```
sudo systemctl status faas001_a
sudo systemctl status faas001_b
```

The output you see will look something like this:

```
● faas001_b.service - FizzBuzz as a Service API prototype - instance A
   Loaded: loaded (/home/chaos/src/examples/killer-
whiles/faas001_a.service; static; vendor preset: enabled)
   Active: inactive (dead)
```

As you can see, the API server instances are loaded but inactive. Let's go ahead and start them both via systemctl by issuing the following commands at the command line:

```
sudo systemctl start faas001_a
sudo systemctl start faas001_b
```

Note that these are configured to respond correctly to only the /api/v1/ endpoint. All other endpoints will return a 404 response code.

Now, onto the next component: the load balancer. The load balancer is an NGINX instance, configured to distribute traffic in a round-robin fashion between the two backend instances, and serve on port 8003. It should model the load balancer from our scenario accurately enough. It has a basic configuration that you can take a sneak peek into by issuing this at your command line:

```
cat ~/src/examples/killer-whiles/nginx.loadbalancer.conf | grep -v "#"
```

You will see the following:

```
    upstream backend {
        server 127.0.0.1:8001 max_fails=1 fail_timeout=1s;
        server 127.0.0.1:8002 max_fails=1 fail_timeout=1s;
    }
    server {
        listen 8003;

        location / {
            proxy_pass http://backend;
            proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        }
    }
```

Configuring NGINX and its best practices are beyond the scope of this book. You won't need to know much more than that the server should behave like the one described in

the scenario at the beginning of the chapter. The only thing worth mentioning might be the `fail_timeout` parameter set to 1 second, which means that after one of the servers returns an error (or doesn't respond), it will be taken away from the pool for 1 second, and then gracefully reintroduced. `max_fails` configures NGINX to consider a single error response enough to take the instance out of the pool. NGINX is configured to listen on port 8003 on localhost in your VM.

Let's make sure the load balancer is also up and running by running this at your command prompt:

```
sudo systemctl start nginx
```

To confirm that you can successfully reach the API servers through the load balancer, feel free to use `curl` to reach the load balancer. You can do that by making an HTTP request to localhost, on port 8003, requesting the only implemented endpoint `/api/v1/`. To do that, run the following command in your prompt:

```
curl 127.0.0.1:8003/api/v1/
```

You should see this amazing response:

```
{
    "FizzBuzz": true
}
```

If that's what you receive, we are good to go. If you're tempted to take a look at the source code now, I'm not going to stop you, but I recommend holding off and looking a bit later. That way, it's easier to think about these components as black boxes with certain behaviors you are interested in. OK, we're done here; it's time to make the system do some work by generating some load!

Pop quiz: Which step is not a part of the chaos experiment template?
Pick one:

1   Observability
2   Steady state
3   Hypothesis
4   Crying in the corner when an experiment fails

See appendix A for answers.

2.4.1   *Ensure observability*

There are many ways to generate HTTP loads. To keep things simple, let's use Apache Bench, preinstalled and accessible through the `ab` command. The usage is straightforward. For example, to run as many requests as you can to your load balancer with

concurrency of 10 (-c 10) during a period of up to 30 seconds (-t 30) or up to 50,000 requests (whichever comes first), while ignoring the content length differences (-l), all you need to do is run this command at your prompt:

```
ab -t 30 -c 10 -l http://127.0.0.1:8003/api/v1/
```

The default output of ab is pretty informative. The bit of information that you are most interested in is Failed requests; you will use that as your success metric. Let's go ahead and take a look at what value it has in the steady state.

2.4.2 Define a steady state

To establish the steady state, or the normal behavior, execute the ab command in your terminal:

```
ab -t 30 -c 10 -l http://127.0.0.1:8003/api/v1/
```

You will see output similar to the following; it is a little verbose, so I removed the irrelevant parts:

```
(...)
Benchmarking 127.0.0.1 (be patient)
(...)
Concurrency Level:      10
Time taken for tests:   22.927 seconds
Complete requests:      50000
Failed requests:        0
(...)
```

As you can see, Failed requests is 0, and your two API servers are serving the load through the load balancer. The throughput itself is nothing to brag about, but since you're running all the components in the same VM anyway, you're going to ignore the performance aspect for the time being. You will use Failed requests as your single metric; it is all you need for now to monitor your steady state. It's time to write down your hypothesis.

2.4.3 Form a hypothesis

As I said before, you expect our system to handle a restart of one of the servers at a time. Your first hypothesis can therefore be written down as follows: "If we kill both instances, one at a time, the users won't receive any error responses from the load balancer." No need to make it any more complex than that; let's run it!

2.4.4 Run the experiment

The scene is now set, and you can go ahead and implement your very first experiment with some basic bash kung fu. You'll use ps to list the processes you're interested in, and then first kill instance A (port 8001), then add a small delay, and then kill

instance B (port 8002), while running `ab` at the same time. I've prepared a simple script for you. Take a look by executing this command at your prompt:

```
cat ~/src/examples/killer-whiles/cereal_killer.sh
```

You will see the following output (shortened for brevity):

```
echo "Killing instance A (port 8001)"
ps auxf | grep 8001 | awk '{system("sudo kill " $2)}'   ◁
(...)

echo "Wait some time in-between killings"
sleep 2
(...)

echo "Killing instance B (port 8002)"
ps auxf | grep 8002 | awk '{system("sudo kill " $2)}'   ◁
```

Searches output of ps for a process with string "8001" (faas001_a) in it and kills it

Waits 2 seconds to give NGINX enough time to detect the instance restarted by systemd

Searches output of ps for a process with string "8002" (faas001_b) in it and kills it

The script first kills one instance, then waits some, and finally kills the other instance. The delay between killing instances is for `nginx` to have enough time to re-add the killed instance A to the pool before you kill instance B. With that, you should be ready to go! You can start the `ab` command in one window by running the following:

```
bash ~/src/examples/killer-whiles/run_ab.sh
```

And in another window, you can start killing the instances by using the `cereal_killer.sh` script you just looked at. To do that, run this command in your prompt:

```
bash ~/src/examples/killer-whiles/cereal_killer.sh
```

You should see something similar to this (I shortened the output by removing some less relevant bits):

```
Listing backend services
(...)

Killing instance A (port 8001)
● faas001_a.service - FizzBuzz as a Service API prototype - instance A
   Loaded: loaded (/home/chaos/src/examples/killer-
whiles/faas001_a.service; static; vendor preset: enabled)
   Active: active (running) since Sat 2019-12-28 21:33:00 UTC; 213ms ago
(...)

Wait some time in-between killings

Killing instance B (port 8002)
● faas001_b.service - FizzBuzz as a Service API prototype - instance B
   Loaded: loaded (/home/chaos/src/examples/killer-
whiles/faas001_b.service; static; vendor preset: enabled)
   Active: active (running) since Sat 2019-12-28 21:33:03 UTC; 260ms ago
(...)
```

```
Listing backend services
(...)

Done here!
```

Both instances are killed and restarted correctly—you can see their process ID (PID) change, and `systemd` reports them as active. In the first window, once finished, you should see no errors:

```
Complete requests:      50000
Failed requests:        0
```

You have successfully confirmed your hypothesis and thus concluded the experiment. Congratulations! You have just designed, implemented, and executed your very first chaos experiment. Give yourself a pat on the back!

It looks like our system can survive a succession of two failures of our API server instances. And it was pretty easy to do, too. You used `ab` to generate a reliable metric, established its normal value range, and then introduced failure in a simple bash script. And while the script is simple by design, I'm expecting that you thought I was being a little trigger-happy with the `kill` command—which brings me to a new concept called *blast radius.*

2.5   *Blast radius*

If you were paying attention, I'm pretty sure you noticed that my previous example `cereal_killer.sh` was a bit reckless. Take a look at the lines with `sudo` in them in our `cereal_killer.sh` script by running this command at the prompt:

```
grep sudo ~/src/examples/killer-whiles/cereal_killer.sh
```

You will see these two lines:

```
ps auxf | grep 8001 | awk '{system("sudo kill " $2)}'
ps auxf | grep 8002 | awk '{system("sudo kill " $2)}'
```

That implementation worked fine in the little test, but if any processes showed up with the string 8001 or 8002 in the output of `ps`, even just having such a PID, they would be killed. Innocent and without trial. Not a great look, and a tough one to explain to your supervisor at the nuclear power plant.

In this particular example, you could do many things to fix that, starting from narrowing your `grep`, to fetching PIDs from `systemd`, to using `systemctl restart` directly. But I just want you to keep this problem at the back of your mind as you go through the rest of the book. To drive the point home, figure 2.5 illustrates three possible blast radiuses, ranging from a broad `grep` from the example before to a more specific one, designed to affect only the targeted process.

That's what blast radius is all about: limiting the number of things our experiments can affect. You will see various examples of techniques used to limit the blast radius as we
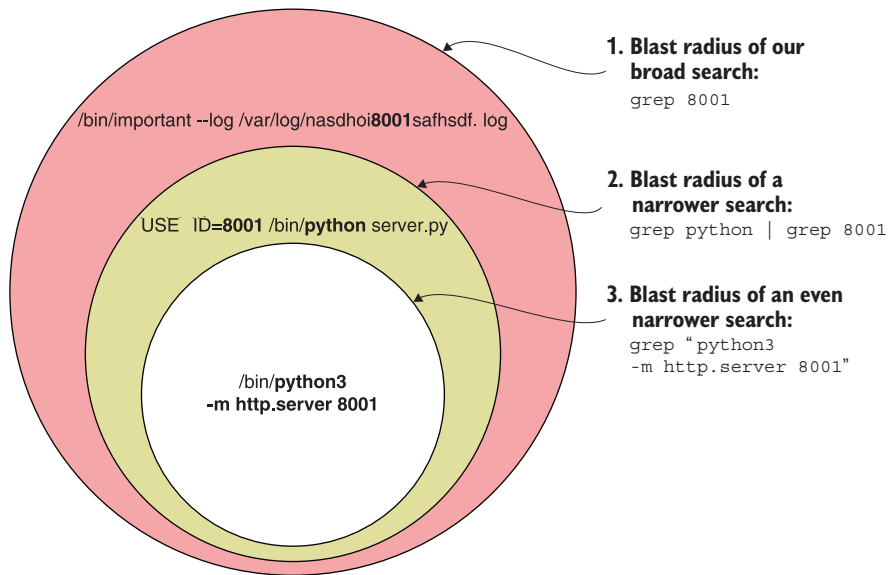
Figure 2.5   Example of blast radiuses

cover various scenarios in the following chapters, but in general they fall into two categories: strategic and implementational.

The preceding situation falls into the latter category of an implementational approach. You can proactively look for ways to make the execution safer, but as with any code, you are bound to make mistakes.

The former category, strategic, is more about planning your experiments in a way so as to minimize the room for catastrophic events if your experiments go awry. Many good software deployment practices will apply. Here are a few examples:

- Roll out the experiment on a small subset of your traffic first, and expand later.
- Roll out the experiment in a quality assurance (QA) environment before going to production (we'll talk about testing in production later).
- Automate early, so that you can reproduce your findings more easily.
- Be careful with randomness; it's a double-edged sword. It can help find things like race conditions, but it might make things difficult to reproduce (we'll also come back to this a bit later).

All right, so knowing your blast radius is important. For this example, we're not going to change the script, but I'd like you to keep the blast radius at the back of your mind from now on. Our first experiment didn't detect any issues, and we've patted ourselves on the back, but wait! The FizzBuzz clients are still seeing errors, which indicates that we didn't go deep enough into the rabbit hole. Let's dig deeper!

> ## Pop quiz: What's a blast radius?
>
> Pick one:
>
> 1 The amount of stuff that can be affected by our actions
> 2 The amount of stuff that we want to damage during a chaos experiment
> 3 The radius, measured in meters, that's a minimal safe distance from coffee being spilled when the person sitting next to you realizes their chaos experiment went wrong and suddenly stands up and flips the table
>
> See appendix A for answers.

2.6   *Digging deeper*

In our first experiment, we have been pretty conservative with our timing, allowing enough time for NGINX to re-add the previously killed server to the pool and gracefully start sending it requests. And by *conservative*, I mean to say that I put the sleep there to show you how a seemingly successful experiment might prove insufficient. Let's try to fix that. What would happen if the API server crashed more than once in succession? Would it continue to work?

Let's tweak our chaos experiment by changing our hypothesis with some concrete numbers: "If we kill an instance A six times in a row, spaced out by 1.25 seconds, and then do the same to instance B, we continue seeing no errors." Yes, these numbers are weirdly specific, and you're about to see why I picked these in just a second!

I wrote a script that does that for you: it's called `killer_while.sh`. Please take a look at the source code by running this in your prompt:

```
cat ~/src/examples/killer-whiles/killer_while.sh
```

You will see the body of the script, just like the following:

```
# restart instance A a few times, spaced out by 1.25 second delays
i="0"
while [ $i -le 5 ]                      ⟵ Introduces a while loop to
do                                        repeat the killing six times
    echo "Killing faas001_a ${i}th time"
    ps auxf | grep killer-whiles | grep python | grep 8001 | awk
    '{system("sudo kill " $2)}'         ⟵ Uses a slightly more conservative
    sleep 1.25                            series of grep commands to narrow
    i=$[$i+1]                             the target processes, and kills them
done

systemctl status faas001_a --no-pager   ⟵ Displays status of the service
                                          faas001_a (--no-pager to prevent
(...)                                     piping the output to less)
```

Sleeps a little bit to give the service enough time to get restarted points to the `sleep 1.25` line.

This is essentially a variation of our previous script `cereal_killer.sh`, this time wrapped in a couple of `while` loops. (Yes, I did use `while` loops instead of `for` loops so that the killer "whiles" joke works. Worth it!).

What do you think will happen when you run it? Let's go ahead and find out by running the script at the command prompt like so:

```
bash ~/src/examples/killer-whiles/killer_while.sh
```

You should see output similar to this (again, shortened to show the most interesting bits):

```
Killing faas001_a 0th time
(...)
Killing faas001_a 5th time
● faas001_a.service - FizzBuzz as a Service API prototype - instance A
   Loaded: loaded (/home/chaos/src/examples/killer-
whiles/faas001_a.service; static; vendor preset: enabled)
   Active: failed (Result: start-limit-hit) since Sat 2019-12-28 22:44:04
UTC; 900ms ago
  Process: 3746 ExecStart=/usr/bin/python3 -m http.server 8001 --directory
/home/chaos/src/examples/killer-whiles/static (code=killed, signal=TERM)
 Main PID: 3746 (code=killed, signal=TERM)

Dec 28 22:44:04 linux systemd[1]: faas001_a.service: Service
RestartSec=100ms expired, scheduling restart.
Dec 28 22:44:04 linux systemd[1]: faas001_a.service: Scheduled restart job,
restart counter is at 6.
Dec 28 22:44:04 linux systemd[1]: Stopped FizzBuzz as a Service API
prototype - instance A.
Dec 28 22:44:04 linux systemd[1]: faas001_a.service: Start request repeated
too quickly.
Dec 28 22:44:04 linux systemd[1]: faas001_a.service: Failed with result
'start-limit-hit'.
Dec 28 22:44:04 linux systemd[1]: Failed to start FizzBuzz as a Service API
prototype - instance A.
Killing faas001_b 0th time
(...)
Killing faas001_b 5th time
● faas001_b.service - FizzBuzz as a Service API prototype - instance B
   Loaded: loaded (/home/chaos/src/examples/killer-
whiles/faas001_b.service; static; vendor preset: enabled)
   Active: failed (Result: start-limit-hit) since Sat 2019-12-28 22:44:12
UTC; 1s ago
  Process: 8864 ExecStart=/usr/bin/python3 -m http.server 8002 --directory
/home/chaos/src/examples/killer-whiles/static (code=killed, signal=TERM)
 Main PID: 8864 (code=killed, signal=TERM)

(...)
```

Not only do you end up with errors, but both of your instances end up being completely dead. How did that happen? It was restarting just fine a minute ago; what went wrong? Let's double-check that you didn't mess something up with the systemd service file. You can see it by running this command in your prompt:

```
cat ~/src/examples/killer-whiles/faas001_a.service
```

You will see this output:

```
[Unit]
Description=FizzBuzz as a Service API prototype - instance A

[Service]
ExecStart=python3 -m http.server 8001 --directory
/home/chaos/src/examples/killer-whiles/static
Restart=always
```

The `Restart=always` part sounds like it should always restart, but it clearly doesn't. Would you like to take a minute to try to figure it out by yourself? Did you notice any clues in the preceding output?

2.6.1 *Saving the world*

As it turns out, the devil is in the details. If you read the logs in the previous section carefully, `systemd` is complaining about the start request being repeated too quickly. From the `systemd` documentation (http://mng.bz/VdMO), you can get more details:

```
DefaultStartLimitIntervalSec=, DefaultStartLimitBurst=
Configure the default unit start rate limiting, as configured per-service
by StartLimitIntervalSec= and StartLimitBurst=. See systemd.service(5) for
details on the per-service settings. DefaultStartLimitIntervalSec= defaults
to 10s. DefaultStartLimitBurst= defaults to 5.
```

Unless `StartLimitIntervalSec` is specified, the default values allow only five restarts within a 10-second moving window and will stop restarting the service if that's ever exceeded. Which is both good news and bad news. Good news, because we're only two lines away from tweaking the `systemd` unit file to make it always restart. Bad, because once we fix it, the API itself might keep crashing, and our friends from Glanden might never fix it, because their clients are no longer complaining!

Let's fix it. Copy and paste the following command at your prompt to add the extra parameter `StartLimitIntervalSec` set to `0` to the service description (or use your favorite text editor to add it):

```
cat >> ~/src/examples/killer-whiles/faas001_a.service <<EOF
[Unit]
StartLimitIntervalSec=0
EOF
```

After that, you need to reload the `systemctl` daemon and start the two services again. You can do this by running the following command:

```
sudo systemctl daemon-reload
sudo systemctl start faas001_a
sudo systemctl start faas001_b
```

You should now be good to go. With this new parameter, instance A will be restarted indefinitely, thus surviving repeated errors, while instance B still fails. To test that, you can now run `killer_while.sh` again by executing the following command at your prompt:

```
bash ~/src/examples/killer-whiles/killer_while.sh
```

You will see output similar to this (again, shortened for brevity):

```
Killing faas001_a 0th time
(...)
Killing faas001_a 5th time
● faas001_a.service - FizzBuzz as a Service API prototype - instance A
   Loaded: loaded (/home/chaos/src/examples/killer-
whiles/faas001_a.service; static; vendor preset: enabled)
   Active: active (running) since Sat 2019-12-28 23:16:39 UTC; 197ms ago
(...)
Killing faas001_b 0th time
(...)
Killing faas001_b 5th time
● faas001_b.service - FizzBuzz as a Service API prototype - instance B
   Loaded: loaded (/home/chaos/src/examples/killer-
whiles/faas001_b.service; static; vendor preset: enabled)
   Active: failed (Result: start-limit-hit) since Sat 2019-12-28 23:16:44
UTC; 383ms ago
  Process: 9347 ExecStart=/usr/bin/python3 -m http.server 8002 --directory
/home/chaos/src/examples/killer-whiles/static (code=killed, signal=TERM)
 Main PID: 9347 (code=killed, signal=TERM)
(...)
```

Instance A now survives the restarts and reports as active, but instance B still fails. You made instance A immune to the condition you've discovered. You have successfully fixed the issue!

   If you fix `faas001_b` the same way and then rerun the experiment with `killer_while.sh`, you will notice that you no longer see any error responses. The order of the universe is restored, and our friends in Glanden can carry on with their lives. You just used chaos engineering to test out the system without looking once into the actual implementation of the API servers, and you found a weakness that's easily fixed. Good job. Now you can pat yourself on the back, and I promise not to ruin that feeling for at least 7.5 minutes! Time for the next challenge!

Summary

- When performing chaos experiments, it's important to be able to observe why a process dies—from a crash, a kill signal, or the OOM Killer.
- The blast radius is the maximum number of things that can be affected by an action or an actor.

- Limiting the blast radius consists of using techniques that minimize the risk associated with running chaos experiments, and is an important aspect of planning the experiments.
- Useful chaos experiments can be implemented with a handful of bash commands, as illustrated in this chapter, by applying the simple four-step template that you saw in chapter 1.

appendix A
Answers to the pop quizzes

This appendix provides answers to the exercises spread throughout the chapters. Correct answers are marked in bold.

Chapter 2

Pick the false statement:

1. Linux processes provide a number that indicates the reason for exiting.
2. Number 0 means successful exit.
3. Number 143 corresponds to `SIGTERM`.
4. **There are 32 possible exit codes.**

What's OOM?:

1. A mechanism regulating the amount of RAM any given process is given
2. **A mechanism that kills processes when the system runs low on resources**
3. A yoga mantra
4. The sound that Linux admins make when they see processes dying

Which step is not a part of the chaos experiment template?

1. Observability
2. Steady state
3. Hypothesis
4. **Crying in the corner when an experiment fails**

What's a blast radius?

1. The amount of stuff that can be affected by our actions
2. **The amount of stuff that we want to damage during a chaos experiment**
3. The radius, measured in meters, that's a minimal safe distance from coffee being spilled when the person sitting next to you realizes their chaos experiment went wrong and suddenly stands up and flips the table

index